# 15-451 Algorithms, Spring 2017

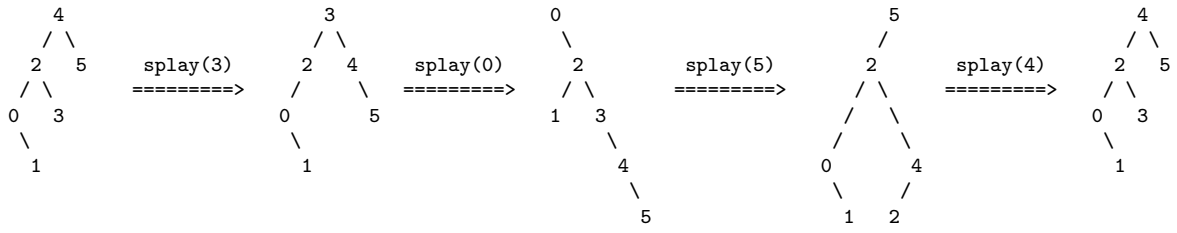
**Homework #2** **Due: February 7–10, 2017**

This is an oral presentation assignment. Again, there are three regular problems (#1-#3) and one programming problem (#4). You should work in groups of three. The sign-up sheet will be online soon (details on Piazza) and your group should sign up for a 1-hour slot. Each person in the group must be able to present every regular problem. The TA/Professor will select who presents which regular problem. You are not required to hand anything in at your presentation, but you may if you choose.

The programming problem will be submitted to autolab, similar to HW#1. You will not have to present anything orally for the programming problem. You can discuss the problem with your group-mates, but must write the program by yourself. Please do not copy.

The bonus problem B2 will be released soon.

(25 pts) 1. **(Cyclic Splaying)** Starting from a tree $T_0$ of $n$ nodes a sequence of $\ell \geq 1$ splay operations is done. It turns out that the initial tree $T_0$ and the final tree $T_\ell$ are the same. Let $k$ be the number of *distinct* nodes splayed in this sequence. (Clearly $k \leq \ell$.) Below is an example where $k = \ell = 4$ and $n = 6$.




(a) Use some setting of node weights to show that the amortized number of splaying steps per operation in this cycle is at most $c_1 + c_2 \log_2 n$. (Smaller $c_1$ and $c_2$ is better.)

(b) Now use a different setting of node weights to show that the average number of splaying steps in this cycle is at most $c_1 + c_2 \log_2 k$. (Smaller $c_1$ and $c_2$ is better.)

(25 pts) 2. **(Looking for Sum One?)** In the Sum-Query problem, you are given two *sorted* arrays $A$ and $B$. Each array has $n$ distinct real numbers. You are also given a real number $q$, and you want to find some indices $i, j \in \{1, 2, \ldots, n\}$ such that $A[i] + B[j] = q$, or else report that no such $i, j$ exist.[1] You are only allowed to use the operation `test(i,j)` that reports back the result of comparing $q$ with $A[i] + B[j]$.

(a) Give an algorithm solving Sum-Query using $O(n)$ calls to `test(.,.)`.

[1] If many different $(A[i], B[j])$ pairs sum to $q$, you are allowed to report any one.

(b) In this part we will show that any deterministic algorithm for SUM-QUERY must make at least $n$ calls. [2]

i. Given some *unsorted* array $C$ of $n$ real numbers (say in the range $[0, M]$) and a real value $q \in [0, M]$, show that the problem of finding some index $k$ such that $C[k] = q$ (or reporting that no such $k$ exists) requires $n$ comparisons between $q$ and elements of $C$ in the worst case.

ii. Given such an array $C$ of $n$ real numbers and a real value $q$, all in the range $[0, M]$, show how to construct two sorted arrays $A$ and $B$ such that

- $C[i] := A[i] + B[n + 1 - i]$, and
- if $i + j < n + 1$, then $A[i] + B[j] < 0$, and
- if $i + j > n + 1$, then $A[i] + B[j] > M$.

This construction should require no comparisons between elements.

iii. Use the above parts to infer a lower bound of $n$ calls to `test(.,.)` for the SUM-QUERY problem.

(25 pts) 3. **(Sometimes You Win, Sometimes You Lose.)** Suppose we have a binary counter such that the cost to increment or decrement the counter is equal to the number of bits that need to be flipped. We saw in class that if the counter begins at 0, and we perform $n$ increments, the amortized cost per increment is just $O(1)$. Equivalently, the total cost to perform all $n$ increments is $O(n)$.

Suppose that we want to be able to both increment *and* decrement the counter.

(a) Show a sequence of $n$ operations allowing both increments and decrements and starting from 0 that, without ever making the counter go negative, costs $\Omega(\log n)$ amortized per operation (i.e., $\Omega(n \log n)$ total cost).

To reduce the cost observed in part (a) we'll consider the following *redundant ternary number system*. A number is represented by a sequence of *trits*, each of which is 0, +1, or −1. The value of the number represented by $t_{k-1}, \ldots, t_0$ (where each $t_i, 0 \leq i \leq k-1$ is a trit) is defined to be

$$\sum_{i=0}^{k-1} t_i 2^i.$$

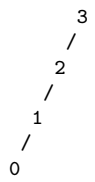
For example, | 1 | 0 | -1 | is a representation for $2^2 - 2^0 = 3$.

The process of incrementing a ternary number is analogous to that operation on binary numbers. You add 1 to the low order trit. If the result is 2, then it is changed to 0, and a carry is propagated to the next trit. This process is repeated until no carry results. Decrementing a number is similar. You subtract 1 from the low order trit. If it becomes -2 then it is replaced by 0, and a borrow is propagated. Note that the same number may have multiple representations (e.g., | 1 | 0 | 1 | = | 1 | 1 | -1 |). That's why this is called a *redundant* ternary number system. The cost of an increment or a decrement is the number of trits that change in the process.

[2] Since arrays $A$ and $B$ are sorted, you cannot just claim that "you have to look at all numbers".

(b) Starting from 0, a sequence of $n$ increments and decrements is done. Give a clear, coherent proof that with this representation, the amortized cost per operation is $O(1)$ (i.e., the total cost for the $n$ operations is $O(n)$). Hint: think about a "bank account" or "potential function" argument.

(25 pts) 4. (**Splaying Depths)** In this problem you will implement splaying. The splay tree will store the keys $0, 1, 2, \ldots, n-1$, in that order. The first line of the input consists of two numbers $n$ and $m$. The initial tree should be a BST with $n$ keys completely unbalanced (to the left); e.g., for $n = 4$ it is




The second line consists of $m$ numbers, which are the keys to be splayed. The first line of your output should have $m$ numbers, giving the depth of the each of the $m$ keys splayed. The second line of the output should be the average depth of these $m$ keys, printed to three decimal places. The input size limits are $1 \leq n, m \leq 10000$. The running time bound will be 10 seconds.

For example if the input is:

```
4 5
0 2 1 1 3
```

Then the output will be:

```
3 3 2 0 2
2.000
```

And if the input is:

```
20 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Then the output will be:

```
19 10 6 3 4 2 3 1 3 1 2 1 4 2 3 1 3 1 2 1
3.600
```